SIMUL_R - A SIMULATION LANGUAGE WITH SPECIAL FEATURES FOR MODEL-SWITCHING AND -ANALYSIS

(THE PREPROCESSORS BAPS, CAPS AND GOMA)

Ronald Ruzicka
TU Vienna, Austria

ABSTRACT

SIMUL_R is a compiler directed simulation language for continous systems. It provides special features for the use of different models for the same system in one program, discrete-events, table-functions, automatic solution of systems with algebraic loops. The runtime-interpreter of SIMUL_R offers several meta-commands, as 'while', 'until', 'for', 'if', macros and subroutines (also recursive). SIMUL_R includes a huge grafic-library (e.g. 3D- and moving graphics). BAPS is a preprocessor for SIMUL_R, which offers the possibility of using bondgraph-models; CAPS provides compartment-models.
GOMA is an optimization-preprocessor for ACSL.

INTRODUCTION

Nearly each system, which is to be analyzed now-a-days, has a very complex structure. Therefore often several models have to be tried and compared afterwards. Comparison of models must in most cases be done by putting one plot over the other and one curve over another. Why not using ***different*** models in ***one*** simulation program?

Very often there occure models of implicit structure and must be solved by hand or by using special statements of simulation languages. Why not simply writing the implicit system as it is articulated and letting solve it by a simulation language?

The upper two arguments are reason enough to construct a new simulation language. Nevertheless it should be compatible with former languages - so the well known CSSL-standard (as for example used in ACSL) can be used.

A third reason for the development of a new language is the manifold of todays problems. Many different sciences use simulation as important tool for the daily work. Therefore a modern simulation language should be ***open***. That means that the user has the possibility of adding new commands to the language.

In SIMUL_R this can be done by simply writing a subroutine, which uses given functions for decoding an input line. These subroutine can be defined by using the programming-language C (which is the 'host'-language for SIMUL_R, as is FORTRAN for ACSL), or as well FORTRAN, or other high-level languages.

THE SIMUL_R-COMPILER

The SIMUL_R-Compiler (let us simply call it 'SIMCO') compiles the source-text of a simulation-program into a C-program, which is linked together with the SIMUL_R runtime-library.

As told by the CSSL-standard such a program is divided into a part of statements, which should be executed before one simulation-run, the system-defining equations and the statements, which have to be executed after the simulation-run.

Differential equations - as matrices and vectors (up to 16 dimensions!), too - can be defined as well as table-functions (with constant or linear interpolation between the breakpoints) or discrete-events. Discrete events can be scheduled once, periodic by time, or when a special case occurs (they also can be scheduled by time during run-time in the runtime-interpreter).

Algebraic Loops

As described above SIMUL_R provides the automatic solution of implicit systems. SIMCO transforms the system with algebraic loops into the equivalent zero-search problem. For example take the equation

x = x*x + t,

which you want to be solved over x, taking t from (1²t²2). You simply write the equation as written above, invoke SIMCO by using the special flag '-a' and start the simulation over (1;2).

Different Models In One Program

A SIMUL_R-sourcetext may contain more than one model; for example a nonlinear and a linear model for the same system (besides: entering the models you can use free format, no more search for the 7^{th} column ...). During runtime you can set the model you just want to be simulated by simple commands (and only this model will be computed). By using the interpreter-loop-commands (called 'meta-commands') simulation-runs with high-level model-switching and -scheduling can be performed.

Within a SIMUL_R-sourcetext also meta-commands (as 'while', 'for', 'if', 'include', macros and meta-variables) may be used. These make it much easier to create complex, but easy-to-read source-codes.

THE SIMUL_R-RUNTIME-INTERPRETER

As it is known the usefullness of a compiler-directed simulation-language mainly depends on the computing-features of its runtime-interpreter. The SIMUL_R-runtime-interpreter provides all normally used commands: start and restart (continue) of simulation-runs (continue ***without*** loss of plot-data!), assignment of algebraic expressions (including functions like exp, log, sin, ...) to variables and display of algebraic expressions.

Loop-Commands

The interpreter also offers the same meta-commands as the compiler. So loops (while, until, for, loop) and dependent computation (if, ifnot), macros and including of files (which can be seen as the calling of a subroutine or the invocation of a batch-file) are possible.

Some usefull commands for the treating of table-functions are supported: simple assignment of one table to another and the so called 'assign'-command. With 'assign' a table can be linked to a variable. During the following simulations the values of the so defined variable (seen over time t) are assigned to the table, better to say: they are recorded. So curves can be saved and used in other models.
Together with the meta-commands these table-function-commands build up a strong tool for complex examinations on dynamic models as e.g. integral-equations, delay-problems, optimization of control-functions.

Optimization, Zero-Search

SIMUL_R is an open system. That means that the user can define his own special commands for

the runtime-interpreter as subroutine of a highlevel programming-language (e.g. C or FORTRAN). Some examples of such commands are part of the SIMUL_R system: parameter-optimization (with parameter-constraints), zero-search, steady-state-analysis, computation of the Jacobian-matrix, computation of eigenvalues and eigenvectors.

A simple example can show how these features work together: take a nonlinear model and a linear model for the same dynamical system as the two models of a SIMUL_R-program. Start with the simulation of the nonlinear model until a special point of time; then invoke the computation of the Jacobian (which writes the matrix-elements to the matrix used in the linear model) and then switch to the linear model. Here you can go on and compare the found solution of your linear-model with the solution of the nonlinear model (which has been recorded in a table-function).

SIMUL_R-Graphics

The graphics-library not only provides 'normal' plots (as known from other simulation languages) with axes (with free selectable scales), curve-marking, etc., but also the possibility of drawing many plots in one picture (for comparisson purposes). Three dimensional plots (e.g. for documentation of different curves during parameter-variation) are supported as well as 'cross'-plots.

'Cross'-plots are drawings, in which not the value of one variable is plotted over the value of another (for example over time) but where the values of several variables are shown simultaneously and connected with lines. Think for example of the discretization of a partial differential equation for the wave-motion of a cord. Such a model contains variables, which at definite points show the distance of the cord from a zero-line. By plotting and connecting the so computed values you can get an impression of the movement of the cord.

This impression can be improved by another feature of the SIMUL_R-graphics-library: it is not necessary to draw all curves (which represent the position of the cord in our example) in one plot, but you can draw them one behind the other - a moving picture. This moving-feature can be used to show the development of the solution of an integral-equation (found with an iterative method using the runtime-interpreter-features of SIMUL_R), too.

There is a third way of plotting moving-pictures: simple curves can be drawn as moving point-tangent-pairs.

A Thread Pendulum

This example shows model-switching in SIMUL_R in the runtime-interpreter. A thread pendulum can be seen as a two-model-system: one model for the phase of the free fall and another for the phase, when the thread has its full length (a circular phase). Fig. 1 shows the SIMUL_R-program, Fig. 2 the runtime-interpreter commands and Fig. 3 the plot y over x.

```
DO {

 "ATAN is necessary to get the right sign"
float ATAN(a)
 double a; {
  if (x==0)
    if (y>=0) return((float)(PI/2));
    else return((float)(0-PI/2));
  if (x<0)
    if (y<0) return((float)(atan(a)-PI));
    else return((float)(0-atan(0-a)-PI));
  else
```

```
    if (y>=0) return((float)(atan(a)));
    else return((float)(0-atan(0-a))); }

};  "DO"


thread_circular {

  CONSTANT m=1, g=9.81, tend=3, r=1;
  CONSTANT phi0=0;
  CONSTANT v0phi = -3, Res=0.4;                    "small resistance"
  EXTERN sin(1), cos(1);

  SE1 = -m*g;

DYNAMIC {

DERIVATIVE {
  phi = INTEG(vphi,phi0);
  x = r*cos(phi);
  vx = r*cos(phi)*vphi;
  y = r*sin(phi);
  vy = r*sin(phi)*vphi;
  L1L2 = cos(phi);
  Resistance = -vphi*Res;
  vphi = INTEG((SE1*L1L2+Resistance)/m,v0phi);

   }  "DERIVATIVE"

  TERMINATE (t>tend || m*g*sin(phi)>m*r*vphi*vphi);
   }  "DYNAMIC"

  vphi_alt=vphi;
   }  "thread_circular"


thread_fall {

      "constants defined in another model"
  CONSTANT EXTERN m, EXTERN g, EXTERN tend, EXTERN r;
  CONSTANT EXTERN phi0;
      "initial values"
  CONSTANT x0 = 1, y0 = 0, p0y = 0;
  CONSTANT PI_2 = 1.570796327;
  LOGICAL logic;
  EXTERN ATAN(1);

  SE1 = -m*g;
  vx = 0;
  vphi_alt=(vy/x0-y0*vx/x0/x0)/(1+y0*y0/x0/x0);


DYNAMIC {

DERIVATIVE {
```

```
 phi = SWI(logic,ATAN(y/x),
             SWI(y<0,(0-(PI_2)),PI_2));
 vphi = SWI(logic,
    (vy/x-y*vx/x/x)/(1+y*y/x/x),vphi_alt);
 x = INTEG(vx,x0);
 y = INTEG(vy,y0);
 vy = INTEG(SE1/m,v0y);
 logic = (x!=0);

  }  "DERIVATIVE"

 vphi_alt=vphi;
 TERMINATE (t>tend || x*x+y*y>r*r);
  }  "DYNAMIC"

  }  "thread_fall"
```

Fig. 1 SIMUL_R-program for thread pendulum

```
akt_mod=thread_circular;
start;
#while t<tend # cont thread_fall;
            cont thread_circular;
#end
```

Fig. 2 Runtime-interpreter commands

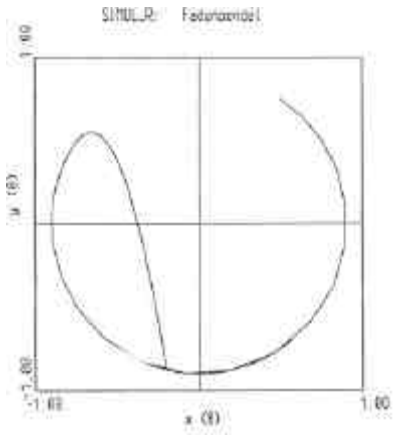

Fig. 3 Plot y over x

BAPS - A BONDGRAPH PREPROCESSOR

Often systems are given not in equations, but implicit in diagrams - as in wiring diagrams or similar drawings in mechanics or thermodynamics. In many cases it is not easy and time-expensive to deduce the equivalent equations from such a diagram for using them within a simulation-environment. Therefore preprocessors has been created to transform these diagrams into a readable form for simulation languages.

BAPS (Bondgraph Analysis and Program Synthesis) is a preprocessor for bondgraphs, which is an universell describing method for electrical, mechanical, thermodynamical and interdisciplinere systems. It translates a bondgraph, which is defined by writing the Elements and the junctions in an easy-to-learn language, into a SIMUL_R-program. Nevertheless it can be translated into ACSL or any other simulation language as well, because BAPS is implemented in a very general way.

The BAPS-System consists of two parts: the 'compiler' and the 'linker'. The compiler reads and analyzes the BAPS-sourcetext and the bondgraph and then generates an intermediate-code. This code is translated into a simulation language by the linker. So only the linker must be changed, if another 'host'-simulation-language is to be used (until now there exist linkers for SIMUL_R, ACSL and HYBSYS, which is a block-oriented language for hybrid simulation).

BAPS not only deals with linear bond-graphs (as defined in (1)), but also with nonlinear bondgraphs. There are several ways of defining nonlinearities:

Elementconstants (as resistance or capacity) can be any algebraic expression; for each element a nonlinear equation may be defined; events can be defined and used as switches in algebraic expressions. These events can also be used to build up tdj-junctions (that are junctions, which change during the course of time - so the topology of a bondgraph can be changed during simulation!) or to generate discontinous states. Table functions can be used, too.

BAPS inspects the bondgraph (including the nonlinearities!) on algebraic loops and assigns their causalities to the bondgraphs (algorithms in (2)). It outputs, if wanted, a list of elements and edges with the assigned causalities.

BAPS supports model-development by giving the possibility of using several models (or bondgraphs) within one BAPS-sourcetext. Not fully specified models can also be used. Within an UNDEFINED model (that is the keyword of BAPS for such models) only input and output variables must be specified; dependencies can be specified in a very abstract manner by saying simply: variables a, ..., b depend on x, ..., y).

Within a BAPS-sourcetext the above introduced 'meta-commands' can be used, too. These for example can be used in asymptotic bondgraph-models. These are models, which approximate asymptotically a real-system (e.g. discretizations, solving of partial differential equations by line-methods).

A Simple Circuit

Here is an example for bondgraph-modelling with BAPS: a simple electrical circuit with a source, a resistor and a capacity. Fig. 4 shows the wiring diagram, Fig. 5 the equivalent bondgraph and Fig. 6 the BAPS-Sourcetext.

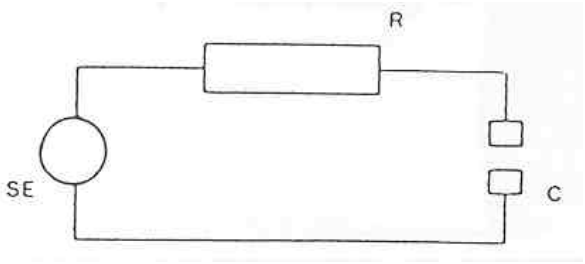

Fig. 4 RC-Circuit

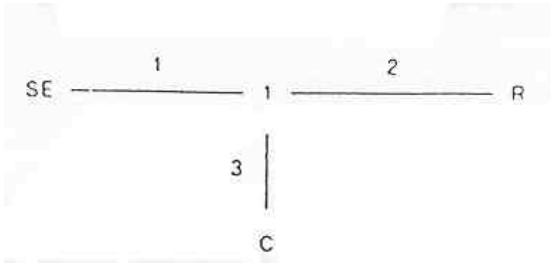

Fig. 5 Bondgraph-description

```
RC_circuit:

    "Bondgraph-description:"

SE 1; 1 1 2 3; R 2; C 3;    {

    "Constant-definition"

CONSTANT TEND=0.05, f = 50*2*3.1416,
       ampl=220;
EXTERN SIN(1), T;

TERMINATE T>=TEND;

    "Elementconstants and -equations"

SE 1: SE1 = ampl*sin(T*f);
R  2: R2  = 100;
C  3: C3  = 0.0001;
     q03 = 0;   "Initial value for state"
}
```

Fig. 6 BAPS-sourcetext

CAPS - A COMPARTEMENT-PREPROCESSOR

CAPS (Compartment Analysis and Program Synthesis, (3)) is a program, which translates a compartment-model-description into the source-text of a simulation language. It supports nonlinear compartement-models, optimization and statistical problems.

Compartments are used to describe biological systems, chemical processes, pharmakokinetics and -dynamics, epidemic spread outs as well as growing of populations.

A spezial feature of CAPS is the opportunity of converting compartment-models into Volterra-systems (these are systems with differential equations with linear right sides, if the logarithmic derivation-operator d(ln)/dt is used).

A CAPS-sourceprogram consists of definitions of constants, tables, compartements, nonlinear-blocks, infusion (idealized adding of substances at a point of time, which leads to discontinouities)

and rates (these describe the flow between compartements). CAPS also supports macros and inluding of files.

A hydrophobia-spreading-out model for foxes

This model (4) can be seen as network of knods, which symbolize regions in a country. Such a knod can be described in a diagram as in Fig. 7 (x is the number of healthy, y of hydrophobia-ill and z the number of died foxes). Fig. 8 shows a macro for a knod in CAPS-syntax. The increase of the number of healthy foxes can be decribed as continouing during the year, or once in spring by defining 'inc1' respectively 'inc2' at the command line of CAPS.

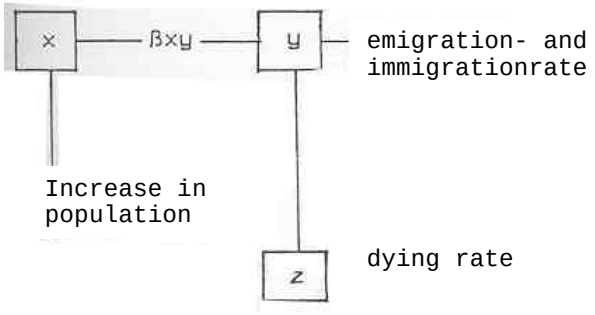

Fig 7. A compartment-system for ill foxes

```
MACRO knod(x,y,xicw,yicw);
CONST x_ic = xicw,
      y_ic = yicw;
COMP x(x_ic), y(y_ic);  "dead foxes are not interesting"
BLOCK inf_x;
inf_x=beta*x*y;
RATE x_out, y_in, y_out;
x_out = x, inf_x, 1;
y_in  = inf_x, y, 1;
y_out = y, EMPTY, gamma;
    "increase of population"
IF inc1 INSERT <inc1.cap>;
IF inc2 INSERT <inc2.cap>;
END;
```

Fig. 8 CAPS-sourcetext

GOMA - AN OPTIMIZING PREPROCESSOR FOR ACSL

In opposite to SIMUL_R ACSL includes no optimizing features. Therefore a preprocessor has been developed at the TU Vienna for solving parameter- and control-function-optimization problems.

GOMA (Generation of Optimizing Models in ACSL (5,6)) reads the optimizing parameters specified by the user and the ACSL-model, adds statements to the model (such as discrete-sections, if control-optimization has been chosen) and generates a new simulation main-program.

The user input contains information on the term to be optimized, number of linear constraints, nonlinear constraints, ending-conditions (for example the difference between a state-variable and its aimed end-value). If control-functions shall be optimized the number of control-functions, the number of breakpoints of the parametrization, the type of the control (constant, linear, cubic-splines, bangbang and user-defined functions); if parameter-optimization has been chosen then the parameters to be optimized must be listed by the user.
All this information can be read from a file or interactive.

GOMA generates an optimizing environment for control-functions, which computes these functions as piece-wise constant or linear functions or as cubic splines with equidistant breakpoints; the function values at these points are optimized. If bangbang or user-defined functions have been chosen, the user supports the functions over intervals and tells in which interval which function must be used; here the length of each interval is optimized.

REFERENCES

1 D. Karnopp, R. Rosenberg, ***System Dynamics: A Unified Approach***, Willey-Interscience, New York, 1975, 402 pages

2 R. Ruzicka, ***Eine Methode zur theoretischen und praktischen Darstellung und Analyse von Bondgraphen unter besonderer Beachtung von Nichtlinearitäten***, Dissertation TU Vienna, 1987, 273 pages

3 A. Sauberer, ***Darstellung, Analyse und Simulation von Kompartmentsystemen unter Berücksichtigung von Nichtlinearitäten***, Dissertation TU Vienna, 1987, 320 pages

4 W. Timischl, "Influence of Landscape on the Spread of an Infection", ***Bull. of Mathematical Biology***, Vol. 46, No. 5/6, Pergamon Press, 1984, pp 869-877

5 M. Gräff, ***Berechnung von optimalen Steuerungen für dynamische Prozesse durch Parameteroptimierung,*** Abt.Bericht 2, TU Vienna, 1986

6 F. Breitenecker, M. Gräff, R. Ruzicka, S. Sauberer, I. Troch, ***Optimierung in ACSL, GOMA - Ein Preprozessor zur automatischen Generierung von Optimierungsprogrammen***, Abt. Bericht 4, TU Vienna, 1987